# 5. Reading and Writing Files

5.1. Introduction

CERN's Large Hadron Collider produces mountains of data: about a gigabyte (10^9 bytes) per second. That's enough to fill a couple of hundred laptop-sized disks per day! This data is saved in files, and these files are distributed around the world for analysis.

Early computers read data from punched cards, or from paper tape with holes punched into it. The pattern of holes on each card was a code that represented numbers or letters. "Keypunch operator" was a job much-advertised in the help-wanted section of the newspaper. A keypunch machine was similar to a typewriter. As the operator typed, holes were punched in the appropriate places on the card. Some keypunch machines also typed the words onto the card, so you could look at it and easily see what was encoded on it (although many programmers became quite adept at reading the holes themselves).

Each punched card could store about eighty bytes of information. If digital cameras had existed at that time, storing a single photo would have required tens of thousands of cards. As computers became faster and capable of dealing with larger data sets, new storage technologies had to be developed. One of these was magnetic media, first in the form of tapes and later disks. Early reel-to-reel tapes of the type introduced by IBM in the 1960s could hold several tens of megabytes: enough for a few photographs from a modern camera. Removable "diskettes" (also called "floppy disks") were developed in the 1970s and 80s. These couldn't hold as much data as tapes, but they were were convenient for storing a few spreadsheets or word-processing documents. "Hard disks", of the type still in use today, can hold several terabytes (10^{12} bytes) of data. That's enough to hold hundreds of thousands of digital photos.
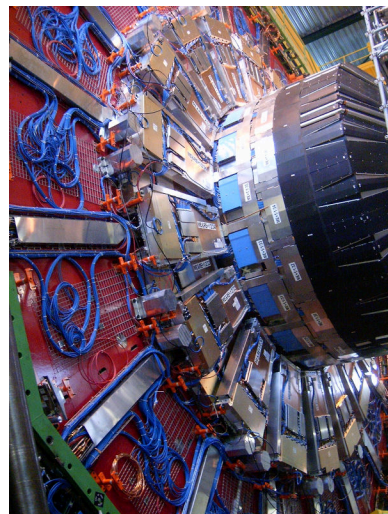
A part of the CMS detector, at CERN's Large Hadron Collider.

Source: Wikimedia Commons

A keypunch machine in the basement of the UNC Physics building. As late as the 1980s, undergraduates would flock there nightly to punch cards for programming projects.

Source: UNC-Chapel Hill Computing History photo collection

A magnetic tape library at the National Oceanographic Data Center.

Source: Wikimedia Commons

As we've learned in earlier chapters, computers store data in the form of ones and zeros. A "file" on a disk is just a collection of ones and zeros, with a name attached to it so we can find it when we need it. In this chapter, we'll learn how to write data to files and read data from files.

5.2. Writing Files

Until now, we've used the `printf` function to send output to the computer's screen. If we want to write things into a file instead, we can use another function named `fprintf` (for "file `printf`"). Before we can do that, though, we have to do a little preliminary work.

Writing to a file isn't quite as simple as writing to the screen. For one thing, we can usually assume that there's a screen to send our output to, but the file might not exist. If it doesn't exist, do we want to create it, or just give the user an error message? If the file exists already, do we want to replace its contents with something new, or do we want to add content after the end of whatever's already there?

We can control all of these options with the `fopen` function. The `fopen` function "opens" a file and makes it ready for reading or writing.

A companion to `fopen` is the `fclose` function. This makes sure that all data has completely been written to a file. Although programs will usually do this for you automatically when they finish running, it's good practice to explicitly use the `fclose` function to "close" a file when you're done with it.

The `fopen` function returns a value that can be used to identify the file you've opened. This identifier is called a "file handle", since it's like a handle by which you can grab the file when you need it.[1] As you'll see, there's a new kind of variable that we use just for storing file handles.

When you use the `fprintf` function to print something into a file, you tell `fprintf` which file to use by giving it a file handle.

Program 5.1 is a very simple example showing how to open a file, write something into it, and then close it. The program writes the words "Hello File!" into a file named `hello.txt`.

A very famous broken file cabinet. This is the cabinet that was broken into in the Watergate Hotel, at the behest of the Nixon administration. It now resides in the Smithsonian's National Museum of American History.

Source: Wikimedia Commons

[1] This identifier is sometimes referred to as a "file descriptor" or "file pointer". These are all the same thing.
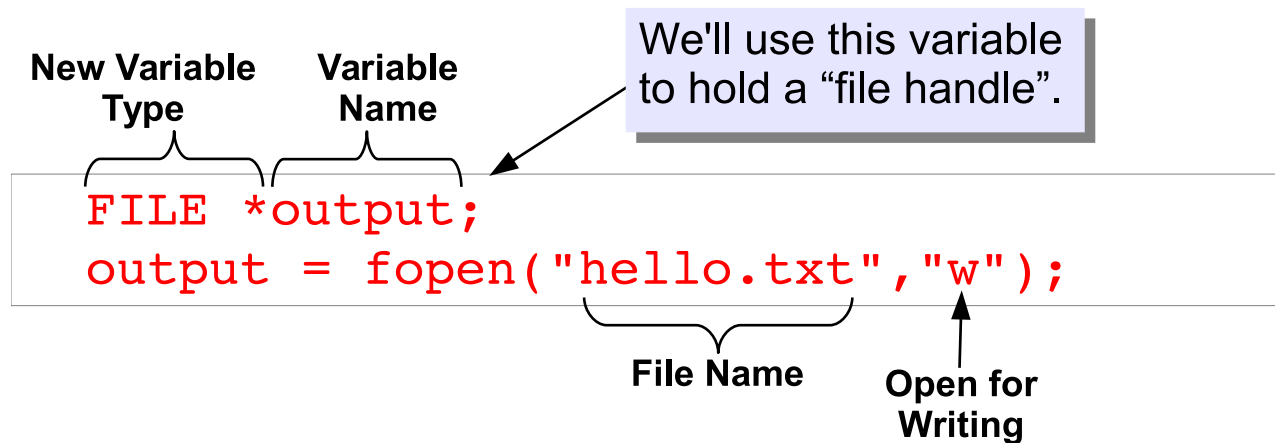
Program 5.1: hellofile.cpp

```cpp
#include <stdio.h>
int main () {
  FILE *output;
  output = fopen("hello.txt","w");

  fprintf( output, "Hello File!\n");

  fclose( output );
}
```

Even though Program 5.1 is short, there's a lot going on in it. Let's look at some of the parts individually. First, let's look a the `fopen` statement:

Figure 5.1: Structure of an `fopen` statement.

As you can see from Figure 5.1, `fopen` takes two arguments: the name of the file to be opened, and a second argument that specifies how we're going to use the file. For example, we can say that we want to read ("r"), write ("w") or append ("a") to the file. There are also other options. See Figure 5.2 for some of them. Usually, you'll only need "r" or "w".
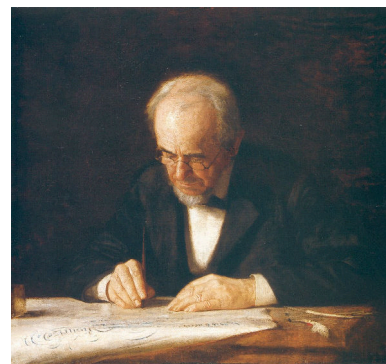
r	Open the file for reading only. Give an error message if the file doesn't exist.
r+	Open the file for reading or writing. Give an error message if the file doesn't exist.
w	Open the file for writing only. If a file with this name already exists, erase it and create a new file.
w+	Open a file for reading or writing. If a file with this name already exists, erase it and create a new file.
a	Open a file for appending (writing at end of file). Create the file if it doesn't exist, but don't erase an existing file.
a+	Open the file for appending and reading. Create the file if it doesn't exist. For existing files, start reading from the top of the file, but write at the bottom.

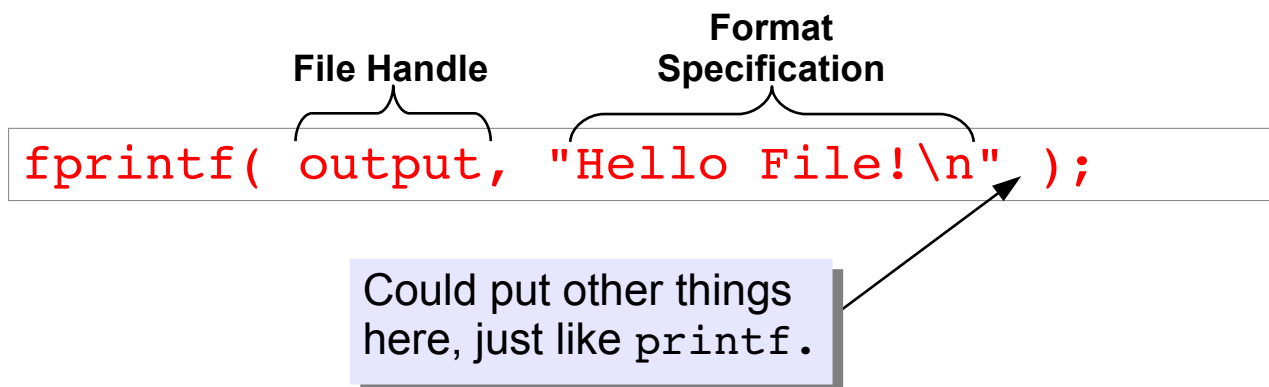Figure 5.2: Various ways that fopen can open a file.

The `fopen` function returns a file handle, which we can capture in a variable for later use. In Program 5.1 we name this variable "`output`", but it can have any name you want to give it. This is a new kind of variable, unlike the `int` and `double` variables we've been using to store numbers. It's a special type of variable just for storing file handles. Just as we might define an integer variable by saying "`int i`", we define this new variable by saying "`FILE *output`". Note the asterisk here is part of the file type. The type of this variable isn't `int` or `double`, it's "`FILE *`".

The Writing Master, by Thomas Eakins.
Source: *Wikimedia Commons*

Once we've stored the file handle in a variable, we can use it to read from a file or write to a file. The `fprintf` function is like `printf`, except that it takes one extra argument: a file handle. In Program 5.1 we use the `fprintf` function to write the text "Hello File!" into the file `hello.txt`, which we've previously opened with `fopen`. We've specified this file by giving `fprintf` the file handle "`output`". If we wanted to, we could open several different files and write different things into each of them. In that case, we'd pick which file we wanted to use by giving the appropriate file handle to the `fprintf` function.

File Handle   **Format Specification**

```
fprintf( output, "Hello File!\n" );
```

Could put other things here, just like `printf`.

Finally, Program 5.1 uses the `fclose` function to make sure everything has been written to the file before the program finishes.

Figure 5.3: Structure of an `fprintf` statement.

Exercise 26: Hello File!

Create, compile and run Program 5.1. When you run the program, you shouldn't see any output since it's being sent into a file instead of to the screen. How can you tell if the program did the right thing?

First of all, look to see if there's a new file. The `ls` command will show you a list of your files. Do you see a file called `hello.txt`?

Next, take a look inside the file by typing `nano hello.txt`. Does it contain the text "Hello File!" as it should?
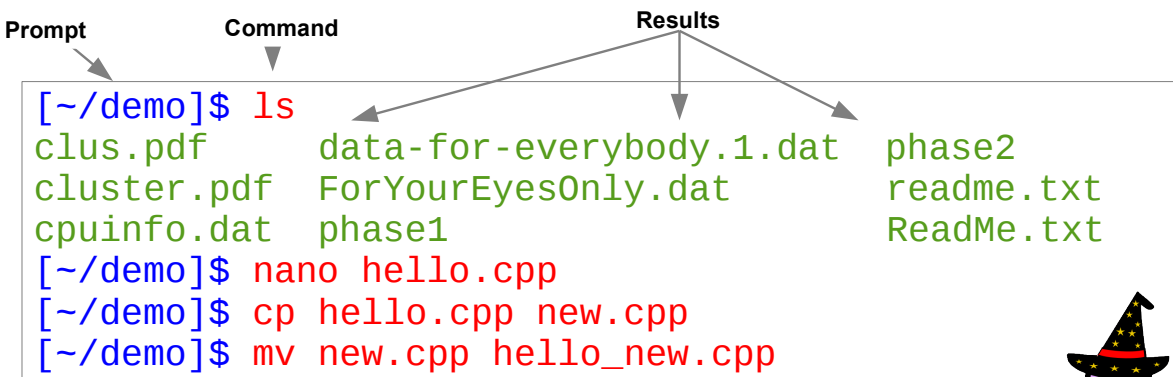
The word "hello" wasn't commonly used until the invention of the telephone. There was initially some disagreement about the proper form of greeting on the new device. Alexander Graham Bell favored "Ahoy!", and some people advocated the jauntier variant "Hoy, Hoy!". Eventually, we settled on "Hello!", and it was so much identified with the device that early telephone operators were referred to as "Hello Girls".

Source: *Wikimedia Commons*

But what about...?

In earlier chapters, we've seen that we can redirect the output of our programs into a file by appending > followed by a file name when we run the program (as we did when plotting the output of our `gutter` program in Chapter 2). You can alternatively use » to append some output at the end of an existing file. For example, you could do the following:

```
./gutter > gutter.dat
./gutter >> gutter.dat
./gutter >> gutter.dat
```

The first command would create a new file called `gutter.dat` and write the program's output into it. The next command would run the program again, and append the output onto the end of the existing file. The last command appends even more output onto the file.

If we can use > or » to redirect a program's output into a file, why would we want to make our C programs write files in any other way? There are at least a couple of reasons:

- Sometimes we want to send some output to the screen and some to a file. Think about a program that asks the user for some input, and then writes out some data. Text that says "Please enter your age" should go to the screen, but we might want the rest of what the program writes to go into a file.

- Sometimes a program needs to write more than one file. Think about a program that sorts data into several categories, and writes each category to a different file. Imagine the program that Santa uses to sort kids into `naughty.dat` and `nice.dat`.

Be careful when using > to send a program's output into a file. If you type the wrong file name, you could accidentally write over a file you want to keep!

5.3. Some Useful Commands for Managing Files

In the exercise above we saw the ls command, and we've been using the commands nano, g++, and gnuplot for a while now. Figure 5.4 summarizes some commands that you might find useful when working with files.

Prompt   Command   Results

```
[~/demo]$ ls
clus.pdf      data-for-everybody.1.dat   phase2
cluster.pdf   ForYourEyesOnly.dat        readme.txt
cpuinfo.dat   phase1                     ReadMe.txt
[~/demo]$ nano hello.cpp
[~/demo]$ cp hello.cpp new.cpp
[~/demo]$ mv new.cpp hello_new.cpp
```

The prompt means "Hello human! I'm ready to receive another command".

Poof!

Some useful commands:

ls	List the contents of a directory.
nano	Edit a file.
cp	Copy a file.
mv	Move (rename, relocate or both) a file.
rm	Delete (remove) a file.
g++	Compile a C (or C++) program.

Figure 5.4: Some useful commands for managing files.

Source: *Openclipart.org*

As we saw in the exercise above, you can use the ls command to show us a list of our files.[2] The cp ("copy") command can be very useful in cases where you want to write a new program that's similar to one you've written in the past. You can make a copy of the old program, with a new name, and then modify the copy as needed.

[2] "ls" is just an abbreviation for "list". As we've seen before, programmers are sometimes lazy typists.

When entering commands at the command line, notice that the computer will usually put a "prompt" at the beginning of each new line. This is some text that might tell you what folder you're working in, or what the computer's name is. The text will vary depending on the type of computer and its configuration. In any case, think of the prompt as the computer's way of saying "OK, I'm ready for you to give me a new

command now."

Although the commands in Figure 5.4 have strange names, you might think of them as wizardly incantations like Harry Potter's "lumos!". By invoking these arcane spells you can cause the computer to do useful things for you.

5.4. Infinite Loops

Sometimes a program doesn't know how much data you're going to give it. Consider Program 5.2 for example.

Program 5.2: input.cpp

```
#include <stdio.h>
int main () {

  int nsiblings;
  int nperson = 0;

  FILE *output;
  output = fopen("siblings.txt","w");

  printf ("Enter the number of siblings, or -1 to quit.\n");

  while ( 1 ) {
    printf ( "Number of siblings for person %d: ", nperson );
    scanf ( "%d", &nsiblings );
    if ( nsiblings < 0 ) {
      break;
    }
    fprintf( output, "%d %d\n", nperson, nsiblings );
    nperson++;
  };

  printf ("Thank you!\n");

  fclose( output );

}
```

Imagine you're collecting data about how many siblings your classmates have. Program 5.2 prompts you to enter the number of siblings each

individual has, and saves the data into a file called `siblings.txt`.

Notice how the program uses the "`while`" loop. As we saw in Chapter 4, a "`while`" loop keeps going for as long as the condition in parentheses is true. Here, the value in parenthesis is just "`1`". Is that true or false?

When a C program comes to a condition statement after an "`if`" or "`while`", the computer converts the condition into a number. If the condition statement is false, the number is zero. Any other number means the statement is true. The "`if`" or "`while`" then uses this number to decide what to do. If we use the number `1` as the condition, it will always be true, so the "`while`" statement in Program 5.2 will keep looping forever unless we somehow tell it to stop. This is called an "infinite loop".

Program 5.2 uses an infinite loop because it doesn't know beforehand how many people you're going to survey. It just keeps asking for more data until you explicitly tell it you're done. When you've collected all of your data, you signify this by giving −1 as the number of siblings. This causes the `break` statement to be acted upon, and the loop terminates.

Infinite loops like this are often used when a program needs to keep doing something until the user tells it to stop. For example, there's an infinite loop underneath the operating system on your computer. It waits for mouse clicks, keystrokes, and other interesting events, and examines them to find out what you're asking it to do. At some point, you may tell the computer to shut down, causing the operating system to clean things up and break the loop.

The address of Apple's corporate headquarters is "1 Infinite Loop".
Source: Wikimedia Commons

Our program assumes that nobody really has a negative number of siblings. How could that even happen? Antimatter??
Source: Wikimedia Commons

Exercise 27: Collecting Data

Create, compile and run Program 5.2. Enter some data from your friends and neighbors, or just make something up. Enter at least ten numbers. When you're done, enter "-1" to stop the program.

Now use *nano* to look at the program's output file: Type "`nano siblings.txt`". Does it look like what you expected?

Exit from *nano*, then start up *gnuplot*. Plot the data you've collected by giving gnuplot the command:

```
plot "siblings.txt" with boxes
```

The result should look something like Figure 5.5. The phrase "`with boxes`" tells gnuplot to draw boxes instead of just plotting points.

Depending on how many points you entered, you may find that *gnuplot* chops off part of the first box. You can fix this by explicitly telling *gnuplot* where you want the *x* axis to start. To do this, type:

```
set xrange [-1:]
```
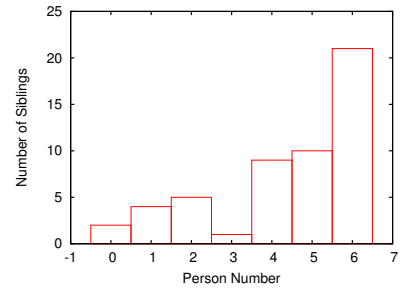
and then type "`replot`".

Figure 5.5: Example sibling data, plotted with *gnuplot*.

5.5. Producing Data Files

Sometimes programs write data, and sometimes they read data. It's often the case that data written by one program will be read by a different program. Think about the experiments at CERN. During an experiment, programs collect the data from particle detectors and write the data into files. Later, perhaps at a university elsewhere, someone uses a different program to read the data files and analyze them.

Let's create a pair of programs that produce and consume data. The first one will write some data into a file, and the second will read the data and do something useful with it. The data will involve a simple physics problem, but don't worry if you don't understand the physics.

Imagine that you fire a gun straight up into the air. The bullet leaves the gun's muzzle at approximately 700 meters per second. As it rises, gravity slows it until eventually it stops rising and begins to fall. Assuming a constant deceleration due to gravity, the velocity of the bullet at any time after it's fired would be:

Figure 5.6: The scenario behind Program 5.3

$$V = V_0 - gt$$

where t is the elapsed time in seconds, V_0 is the bullet's initial velocity, in meters per second, and g is the acceleration due to gravity near the earth's surface, which is about 9.8 m/s^2. Because of the minus sign, the bullet's velocity gets smaller and smaller as time passes, until it

eventually reaches zero, and then it becomes negative (meaning that the bullet has started falling back to earth).

The height of the bullet at any time will be:

$$h = V_0 t - \frac{1}{2} g t^2$$

if we assume that the bullet starts from a height of zero.

Program 5.3 calculates the bullet's velocity and height once per second during the first one hundred seconds of its flight, and writes those values into a file for later analysis.

Program 5.3: bullet.cpp

```
#include <stdio.h>
#include <math.h>
int main () {

  int i;
  double t = 0.0;
  double v;
  double h;
  double v0; // meters per second.
  double delta_t = 1.0; // seconds.
  double g = 9.8; // meters/second.
  FILE *output;

  printf ( "Enter initial velocity (m/s): " );
  scanf ( "%lf", &v0 );

  output = fopen("bullet.txt","w");

  for ( i=0; i<100; i++ ) {
    v = v0 - g*t;
    h = v0*t - 0.5*g*pow(t,2);
    fprintf( output, "%lf %lf %lf\n", t, v, h );
    t += delta_t;
  };

  fclose( output );

}
```

Notice that we've added some comments beside the definitions of our variables to remind us what units we're using. Comments like this can be very helpful if someone else needs to understand your program.

We assume that the acceleration of gravity is a constant, which is approximately true if we don't get too far from the surface of the earth. In Georges Melies' 1902 film *Le Voyage dans la Lune* six men are fired to the moon inside a large artillery shell. Needless to say, our approximation would not hold true in this situation.
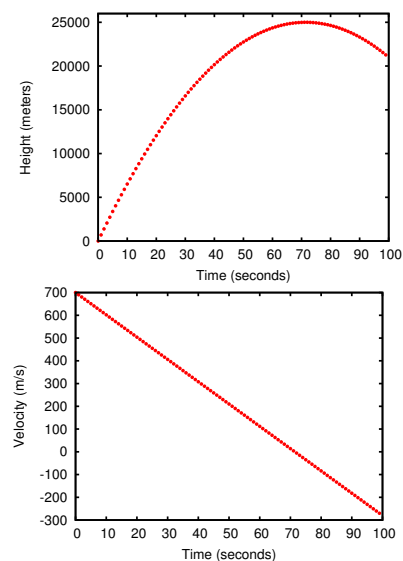
Figure 5.7: A bullet's height and velocity as a function of time, for a starting velocity of 700 m/s.

Exercise 28: Fire At Will!

Create, compile and run Program 5.3. It should ask you for an initial velocity. Use 700 m/s. After the program finishes, use the "ls" command to check that the output file, bullet.txt, has been created. Take a look inside the file with *nano* by typing "nano bullet.txt". There should be three columns of data, for time, velocity, and height.

Now exit from *nano* and use *gnuplot* to plot the bullet's height versus elapsed time, by giving *gnuplot* this command:

```
plot "bullet.txt" using 1:3
```

You should see a graph that looks like top graph in Figure 5.7. Try to identify the bullet's maximum height, and the time at which it reaches this height.

If you have time, you can also graph the bullet's velocity as a function of time by giving gnuplot this command:

```
plot "bullet.txt" using 1:2
```

But what about...?

Notice that Program 5.3 only tracks the bullet for one hundred seconds. The bullet may not reach the ground during that time. What if we wanted the program to track the bullet for as long as it's in the air, and stop when it hits the ground? We could modify the program by replacing the "for" loop with a "do-while" loop, like this:

```
do {
    v = v0 - g*t;
    h = v0*t - 0.5*g*pow(t,2);
    fprintf( output, "%lf %lf %lf\n", t, v, h );
    t += delta_t;
} while ( h >= 0.0 );
```

5.6. Analyzing a Data File

In the exercise above, you might have found that it was hard to tell exactly where the bullet reached its maximum height by looking at the graph of our data. Analyzing data by hand is tedious and imprecise.

Imagine how much harder it is to analyze the data from a huge experiment like the ones at CERN, where billions of data points are recorded per second!
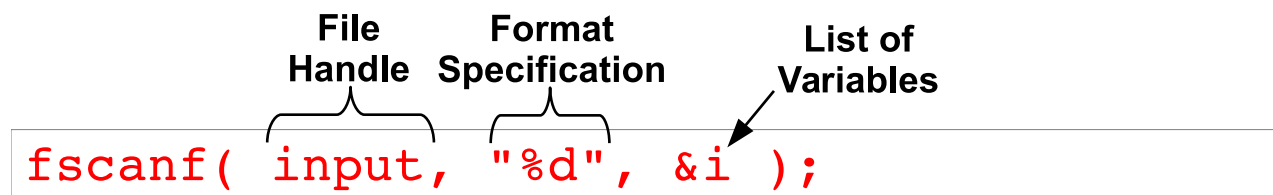
Even for small experiments, it's often necessary to write computer programs to help us analyze data. Let's write a program that can read the bullet program's output file and find the maximum height for us.

Take a look at Program 5.4 on Page 147. This program does several new things. First of all, it opens the file for reading, instead of writing, by giving an "r" to the `fopen` function.

Next, notice that Program 5.4 uses an infinite loop (see the "`while (1)`") to read data from the file. This allows the program to read a file of any length. If we modified our bullet program so that it produced more or fewer lines of data, Program 5.4 would still be able to read the output file.[3]

[3] This would be very important if we changed the loop in our bullet program to a "`do-while`" loop, as in shown in the box above. In that case, we'd never know how many lines of data the program would generate.

Each time Program 5.4 goes around its loop, it reads a line from the `bullet.txt` data file. To do the reading, we use a new function: `fscanf`. The `fscanf` function is like `scanf`, except that it reads data from a file instead of from the keyboard. The first argument we give `fscanf` is a file handle. This tells `fscanf` which file we want to read from. In principle, we could open up several different files and choose which one we want to read by giving the appropriate file handle to `fscanf`. Figure 5.8 shows the structure of a typical `fscanf` statement.


```
fscanf( input, "%d", &i );
```

Just like `scanf`, you should always put an ampersand (`&`) in front of the variable names whenever you read numbers with `fscanf`, and you should avoid "`\n`" in the format specification you give `fscanf`.[4]

Figure 5.8: Structure of an `fscanf` statement.

[4] See Chapter 3.

Since the program uses an infinite loop, we have to do some sort of test inside the loop to see if we're done yet. In this case, we let `fscanf` tell us when there's nothing left to read. Each time we call `fscanf` it returns an integer value that indicates its "status". For example, the returned value may indicate that some error has occurred. One of the values that can be returned is "End Of File". The `#include`

file `stdio.h` defines a special symbol for this: `EOF`.[5] When `fscanf` returns the value `EOF`, that means that we've read all the way to the end of the file and there's nothing left to read. Program 5.4 keeps reading lines until `fscanf` says that it's reached the end of the data file.

Now that we understand the mechanics of reading a file, how do we find the maximum height in our bullet data? First, we create variable called `hmax`, in which we'll store the maximum height. After opening our data file, we read it, one line at a time. Each line of the file contains three numbers: the elapsed time since the bullet was shot, the current velocity, and the current height. We initially set `hmax` equal to the first height value in the file, then each time we read another line from the data file, we look to see if its height is greater than `hmax`. If it is, we make this height the new value of `hmax`. When we're done looking at all of the data, `hmax` should contain the maximum height value.

The program also finds the time at which the maximum height is reached. Whenever the program sets a new `hmax` value, it also sets the variable `tmax` equal to the time value that appears on the same line of the data file. When the program finishes, `tmax` should contain the time at which the maximum height was reached.

Exercise 29: Finding the Maximum

Create, compile and run Program 5.4. Does it give you results that match your expectations?

Now try running your earlier `bullet` program again, this time giving it a different initial velocity, say 600 m/s instead of the 700 m/s you used earlier. Run your `readbullet` program again to find the new maximum height.

If you pick an initial velocity much higher than 700 m/s, you'll find that your `readbullet` program will always tell you that the time at maximum height is 100 seconds. This is because our `bullet` program only tracks the bullet for 100 seconds, and if its initial velocity is too large the bullet will still be rising at the end of this time.

If you have time, look at the new `bullet.txt` file with gnuplot, as you did before, to see if the maximum height looks like it matches the output of `readbullet`.

[5] The status returned by `fscanf` is really just an integer, but stdio.h defines EOF because that's easier to remember. There's no guarantee that different C compilers will return the same number, but they'll all have a stdio.h that defines EOF appropriately for that particular compiler.
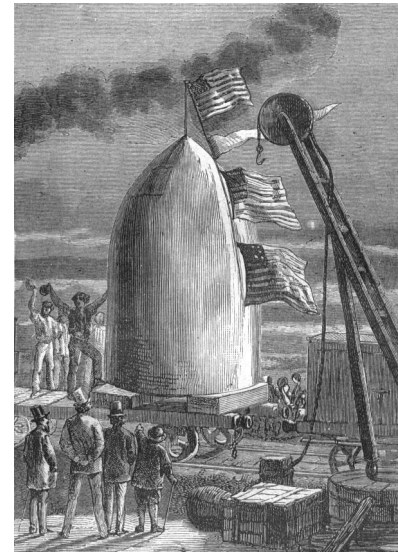
Another group of intrepid adventurers who journeyed to the Moon inside an artillery shell. These are from Jules Verne's *From the Earth to the Moon*, as illustrated by Henri de Montaut.

Source: Wikimedia Commons

Program 5.4: readbullet.cpp

```c
#include <stdio.h>
int main () {
  double t;
  double v;
  double h;

  double hmax;
  double tmax;
  int initialized = 0;

  FILE *input;

  input = fopen("bullet.txt","r");

  while ( fscanf( input, "%lf %lf %lf", &t, &v, &h ) != EOF ) {
    if ( !initialized || h > hmax ) {
      hmax = h;
      tmax = t;
      initialized = 1;
    }
  }

  printf ( "Maximum altitude of %lf after %lf seconds\n", hmax, tmax );

  fclose( input );
}
```

Have we initialized `hmax`?

Open the file for reading, using `"r"`

Stop when we get to the end of the file

Have we found a greater height? (Or do we need to initialize `hmax`?)

`hmax` has now been initialized.

Read lines from the file

5.7. The Perils of Excessive `open/close`

We saw in Chapter 4 that modern computers are very fast. Adding up the square roots of a billion numbers takes only seconds. But some things take longer than others. In particular, it takes a computer a relatively long time to open or close a file.

We can test this with a program like Program 5.5. Here we have a loop that opens and closes a file a million times. Each time around the loop, the program opens the file, writes some text into it, and closes the file. Before the loop starts, the program saves the current time in the variable `tstart`. After the loop finishes, we calculate how much time has passed since `tstart`. The program prints the total time, in seconds, and also prints the time per open/close.

If your computer has an old-fashioned spinning disk this program might take a few minutes to run, with each open/close taking about a millisecond. On a modern solid-state disk each open/close might only take a tenth of a millisecond, but the program will still take several seconds to run. If we increased `ntimes` to a billion, the program would take a thousand times longer (several hours at least). Compare that with the few seconds it took our earlier test program (Program 4.1) to add up the square roots of a billion numbers. You can see that opening and closing files is much slower than just doing math.

Eeek!

Source: Wikimedia Commons

Program 5.5: openclose.cpp

```
#include <stdio.h>
#include <time.h>
int main () {
  int i;
  int ntimes = 100000;
  int tstart;
  double delay;
  FILE * output;

  tstart = time(NULL);

  for ( i=0; i<ntimes; i++ ) {
    output = fopen( "openclose.dat", "w" );
    fprintf( output, "Testing...\n" );
    fclose( output );
  }

  delay = time(NULL) - tstart;
  printf ("Time to open/close %d times: %lf seconds\n", ntimes, delay );
  printf ("Time per open/close: %lf seconds\n", delay/ntimes );
}
```

Exercise 30: Open for Business?

Create, compile and run Program 5.5. How fast is your computer's disk? Remember that on slower disks it can take several minutes for the program to run. If you get tired of waiting, you can stop the program by pressing Ctrl-C.

The lesson we should learn from this is that it's a good idea to avoid unnecessarily opening or closing files. If you write a simulation program like gutter.cpp in Chapter 2 and make the program write its output into a file, it's best to open the output file once, before starting any loops, and then close the file after all the loops are finished. Even though, in principle, you could open the file each time you want to write a new number, that would make your program much, much slower.

Notice that in Program 5.5 we opened the file for writing by giving a "w" as the second argument to fopen. Remember that this wipes out any already-existing file that has the same name. That's why only one small file, containing just the text "Testing", is created when the program is run. The program actually creates and overwrites this file a million times.

Accidentally overwiting an output file is a common programming error. Consider Program 5.6.

Program 5.6: overwrite-test.cpp

```
#include <stdio.h>
int main () {
  FILE *output;
  int i;

  for ( i=0; i<10; i++ ) {
    output = fopen("overwrite-test.dat","w");
    fprintf( output, "%d\n", i);
    fclose( output );
  }

}
```

This program has a loop that sets i to each value from 0 to 9 and writes that value into the output file. If the programmer ran this program he or she might be surprised to find that the output file ends up with

only a single number in it: "9". That happened because the `fopen` and `fclose` statements are inside the loop, and because we gave `fopen` `"w"` (for "write") as its second argument instead of `"a"` (for "append"). We could fix the program by just moving `fopen` and `fclose` outside the loop, like this:

Come in, we're open!
Source: Wikimedia Commons

Program 5.7: overwrite-test.cpp, Fixed!

```c
#include <stdio.h>
int main () {
  FILE *output;
  int i;

  output = fopen("overwrite-test.dat","w");
  for ( i=0; i<10; i++ ) {
    fprintf( output, "%d\n", i);
  }
  fclose( output );

}
```

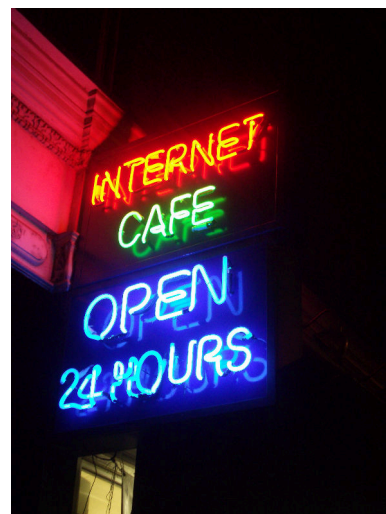Now the program's output file will look like this:

```
0
1
2
3
4
5
6
7
8
9
```

which is probably what the programmer intended.

Closing a file before the program is done with it is another common programming error. If the program above had left `fclose` inside the loop, then the output file would be closed after the first number was written to it. The next time the program tried writing into the file we'd get lots of ugly errors like this:

Error in `./overwrite-test': double free or corruption

This isn't very informative, but the computer is trying to tell us that we're attempting to write into a file that is no longer open.

5.8. Analyzing Other People's Data

Imagine that you're an astronomer, and you've been given the task of analyzing some data about the stars in our local neighborhood. In 1957 astronomer Wilhelm Gliese published the first edition of his list (or "catalog") of nearby stars. It contained entries for about 900 stars. By "nearby", he meant stars within about 65 light-years of Earth. Several editions later, the Gliese catalog now contains about 3,800 stars. The catalog contains information about each star's position, brightness, and color, among other things.

These stars might seem special because they're our closest neighbors. If we were ever to venture into interstellar space, these are the first places we'd visit. You've probably heard of some of them. Sirius, the "Dog Star", is the brightest star in our sky. Tau Ceti and Epsilon Eridani are two nearby Sun-like stars that figure prominently in Science Fiction.

But how close is the nearest star (other than the Sun) to us? Let's write a program to analyze some data about nearby stars and find out.

Program 5.8 reads a file containing x, y, and z coordinates (measured in parsecs[6]) for the position in space of each star. In our `readbullet` program, we analyzed some data to find the maximum value. Here we want to find the minumum value: the star that's closest to earth.

[6] One parsec equals approximately 3.26 light years.

In this data's coordinate system, our Sun is at the origin. If we're given the coordinates of another star, we can find its distance from the Sun like this:

$$r = \sqrt{x^2 + y^2 + z^2}$$

where r is the distance.
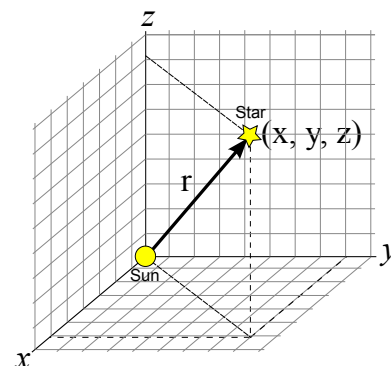
Figure 5.9: Calculating the distance from the sun to another star.
Source: Wikimedia Commons

Program 5.8 reads a star's coordinates from the data file `stars.dat`, then calculates the distance to that star. If that distance is less than the smallest distance we've encountered so far, the program uses it as the new value for the variable `rmin`. Compare this program with Program 5.4, which found a maximum.
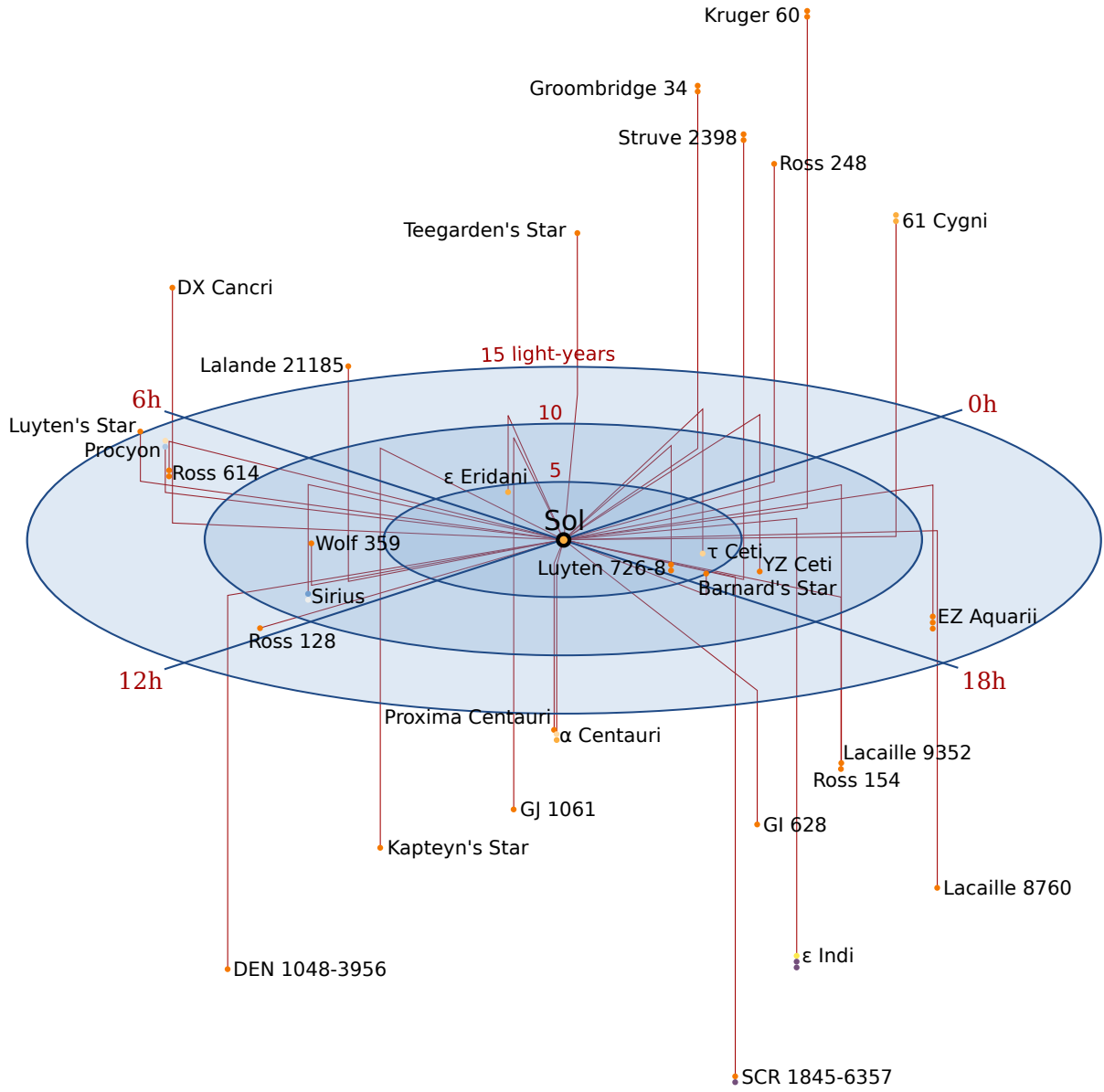
Figure 5.10: The stars in our immediate neighborhood.

Program 5.8: stars.cpp

```
#include <stdio.h>
#include <math.h>
int main () {

  double x;
  double y;
  double z;
  double r;
  double rmin;
  int initialized = 0;

  FILE *input;

  input = fopen("stars.dat","r");

  // Read coordinates for the stars:
  while ( fscanf( input, "%lf %lf %lf", &x, &y, &z ) != EOF ) {
    r = sqrt( x*x + y*y + z*z );
    if ( !initialized || r < rmin ) {
      rmin = r;
      initialized = 1;
    }
  }

  printf ( "Minimum distance is %lf parsecs\n", rmin );

  fclose( input );
}
```

Exercise 31: Seeing Stars

For this exercise you'll need a copy of the data file named
stars.dat. You can find instructions for obtaining it in
Appendix C.1 on page 541. After you have the data file,
create, compile and run Program 5.8. What's the distance
to the closest star in this data set? Its name is Proxima
Centauri.

If you have time, start up *gnuplot* and give it the following
commands (note that the last command is **s**plot, not plot):

```
set xrange [-5:5]
set yrange [-5:5]
set zrange [-5:5]
splot "stars.dat"
```

This should show you a 3-dimensional view of the stars
within about 15 light years from earth. Depending on the
version of *gnuplot* you're using, you may be able to grab this
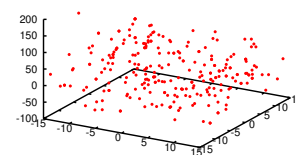plot with the mouse and drag it around to rotate it.

Figure 5.11: Some local stars, plotted
with *gnuplot*.

5.9. Combining Files

Sometimes it's useful to be able to combine data from two or more files
into one. Here are a few techniques for doing that.

Appending:

Imagine you're a teacher. You begin the semester by creating a file
named `grades.dat` that will hold your students' grades. The format
of the file will be one line per student, with the student's ID number
at the beginning of the line, followed by a list of homework grades
separated by spaces. The file might look like Figure 5.12.

After you've created this file, you find that your class is very popular
but the classroom is small. You'll have to teach two groups of students
at different times. To accommodate the second group of students, you
create a new file `grades2.dat` with the same format as the first file.

As the semester goes along, you realize that you'd really like to have one
file that contains all the grades for both sets of students. No problem!
This is a programming class, so you know how to write a program for
combining the two files.

You decide that you just want to append the data from `grades2.dat`
onto the bottom of `grades.dat`, and then ignore `grades2.dat` from
now on. To accomplish this, you write Program 5.9.

```
1 95.0 89.5 100.0
2 79.5 88.0 90.0
3 82.5 87.5 95.5
4 99.0 100.0 97.5
5 88.0 89.0 91.5
6 92.0 93.5 96.0
7 100.0 99.0 95.5
8 90.0 92.0 95.0
9 88.5 92.5 95.0
10 100.0 96.5 90.0
```

Figure 5.12: Your `grades.dat` file
might look like this. Each line begins
with the student's ID number. After
that comes a list of that student's
homework grades.

Program 5.9: append.cpp

```cpp
#include <stdio.h>
int main () {
  FILE *file1;
  FILE *file2;
  int id;
  double h1,h2,h3;

  file1 = fopen("grades.dat","a");
  file2 = fopen("grades2.dat", "r");

  while ( fscanf(  file2 , "%d %lf %lf %lf", &id, &h1, &h2, &h3 ) != EOF ) {
    fprintf (  file1 , "%d %lf %lf %lf\n", id, h1, h2, h3 );
  }

  fclose ( file1 );
  fclose ( file2 );
}
```

Open `grades.dat`
for *appending* by
specifying `"a"`.

Read from `file2`
(`grades2.dat`).

Write to `file1`
(`grades.dat`).

Program 5.9 reads each line of grades2.dat and writes it at the end
of grades.dat. It's written at the end because we told fopen to open
the file for appending, by specifying "a". After running this program,
all of the grades would be in grades.dat.

This program shows that you can have more than one file open at a
time. When we read or write, we specify which file to use by giving
the appropriate file handle to fscanf or fprintf.

Concatenating:

Thinking about your class a little more, it might occur to you that it
would be better to leave both grades.dat and grades2.dat as they
are (since these are important student records!) and create a third,
new file named homework.dat that combines the data from both the
original files. You could write another program (Program 5.10) to do
that.

Program 5.10: concat.cpp

```
#include <stdio.h>
int main () {
  FILE *file1;
  FILE *file2;
  FILE *homework;
  int id;
  double h1,h2,h3;

  homework = fopen("homework.dat", "w");

   file1 = fopen("grades.dat","r");
  while ( fscanf( file1, "%d %lf %lf %lf", &id, &h1, &h2, &h3 ) != EOF ) {
    fprintf ( homework, "%d %lf %lf %lf\n", id, h1, h2, h3 );
  }
   fclose ( file1 );

   file2 = fopen("grades2.dat", "r");
  while ( fscanf( file2, "%d %lf %lf %lf", &id, &h1, &h2, &h3 ) != EOF ) {
    fprintf ( homework, "%d %lf %lf %lf\n", id, h1, h2, h3 );
  }
   fclose ( file2 );

   fclose( homework );
}
```

> Open the new file
> homework.dat for
> *writing* by specifying "w".

> Read data from grades.dat
> and write it to homework.dat.

> Now read data from grades2.dat
> and write it to homework.dat.

grades.dat

grades2.dat

As you can see, Program 5.10 creates a new file named `homework.dat` by giving `fopen` a `"w"`. The program then has two sections: first it reads data from `grades.dat` and writes that data into `homework.dat`. Then it does the same for `grades2.dat`.

Merging:

All is well until the end of the semester. You've graded all of the homework assignments and put the grades into `homework.dat`. You've also graded some quizzes and put those grades into `quizzes.dat`. There were three homework assignments and two quizzes (it was a short course). Each student has one line in each file. Figure 5.13 shows what the `quizzes.dat` file might look like.

Hmmm. It would be really nice if we could combine `homework.dat` and `quizzes.dat` and create a new file that had all of each student's grades, homework and quizzes, on a single line. To do that, you could write something like Program 5.11.

Program 5.11 creates a new file named `allgrades.dat` that will contain one line per student, with all of that student's grades (homework and quizzes). Each line begins with the student' ID number. The new file might look like Figure 5.14.

Notice that the program reads one line from each input file each time it goes around the `while` loop. The `fscanf` statements for reading `homework.dat` and `quizzes.dat` are different, because the files have different formats. Both begin with the student ID number, but there are three homework grades and only two quizzes.

The loop stops (by using the `break` statement) when it reaches the end of either input file. It's important to check both files, to help us deal with mistakes we might have made when we entered the grades. What if we've left a student out of one of the files? In that case the input files wouldn't both be the same length.

Similarly, we put the student ID number into `id1` when we read it from `homework.dat` and we put the number into `id2` when we read it from `quizzes.dat`. If we haven't made any mistakes in creating the input files, these two ID numbers should always match. If they don't, the program gives us an error message telling us so.

Finally, once the program has successfully read a line of homework

```
1 100.0 96.5
2 88.5 92.5
3 90.0 92.0
4 100.0 99.0
5 92.0 93.5
6 88.0 89.0
7 99.0 100.0
8 82.5 87.5
9 79.5 88.0
10 95.0 89.5
```

Figure 5.13: The `quizzes.dat` file might look like this, with each line containing a student's ID number and two quiz grades.

```
1 95.0 89.5 100.0 100.0 96.5
2 79.5 88.0 90.0 88.5 92.5
3 82.5 87.5 95.5 90.0 92.0
4 99.0 100.0 97.5 100.0 99.0
5 88.0 89.0 91.5 92.0 93.5
6 92.0 93.5 96.0 88.0 89.0
7 100.0 99.0 95.5 99.0 100.0
8 90.0 92.0 95.0 82.5 87.5
9 88.5 92.5 95.0 79.5 88.0
10 100.0 96.5 90.0 95.0 89.5
```

Figure 5.14: The file `allgrades.dat`, produced by Program 5.11, might look like this. Each line has the students ID number, followed by three homework grades and two quiz grades.

data and a line of quiz data, it writes all of the data out on a single line of the output file. Notice that the first `fprintf` statement doesn't end with a "\n". Instead, it ends with a space. The next `fprintf` statement picks up where the first one left off, adding more stuff to the end of the same line, and then finishing with a "\n".

Program 5.11: merge.cpp

```c
#include <stdio.h>
int main () {
  FILE *file1;
  FILE *file2;
  FILE *combined;
  int id1, id2;
  double h1,h2,h3;
  double q1,q2;

  combined = fopen("allgrades.dat", "w");

  file1 = fopen("homework.dat","r");
  file2 = fopen("quizzes.dat", "r");

  while (1) {
    if ( fscanf( file1, "%d %lf %lf %lf", &id1, &h1, &h2, &h3 ) == EOF ) {
      break;
    }

    if ( fscanf( file2, "%d %lf %lf", &id2, &q1, &q2 ) == EOF ) {
      break;
    }

    if ( id1 == id2 ) {
      fprintf ( combined, "%d %lf %lf %lf ", id1, h1, h2, h3 );
      fprintf ( combined, "%lf %lf\n", q1, q2);
    } else {
      printf ( "Error! IDs don't match: %d and %d\n", id1, id2);
    }
  }

  fclose ( file1 );
  fclose ( file2 );

  fclose( combined );
}
```

Output file

Input files

Read home-work

Stop when we reach the end of either input file.

Read quizzes

Check to make sure the student IDs from both files match.

Write homework and quiz data on one line.

5.10. Conclusion

In this chapter we've covered the basics of reading from files and writing to files. These same techniques can be used for any numerical data that's stored in the form of multi-column, readable numbers. Programs like *gnuplot* read data files in a way very similar to this. Multi-column numerical data is very commonly used for small-to-moderate sized data sets, although sometimes the columns are separated by commas, colons or other characters besides spaces.[7]

[7] Large data sets are generally stored differently, in formats not readable by humans but which allow the files to be smaller, faster to read, and easier to search. We'll take a look at reading and writing this kind of files later on.

Figure 5.15: In the days before files were stored on disks, students delivered stacks of punched cards to counters like this one in the the basement of the UNC Physics building. Computer operators loaded the stacks into readers, and the program's output was printed (sometimes hours later) and dropped by the operator into a bin, until the student came by to pick it up.

Source: UNC-Chapel Hill Computing History photo collection

Practice Problems

1. Write the following two programs:

   (a) Modify Program 5.3 (the `bullet.cpp` program) so that it writes comma-separated columns into its output file, instead of space-separated columns. Run the program to generate a new `bullet.txt` output file.

   (b) Modify Program 5.4 (the `readbullet.cpp` program) so that it will read the new comma-separated data file.

2. Using *nano*, create a data file called `numbers.dat` that contains a column of at least ten integers (positive or negative), like this:

   ```
   27
   -3
   189
   43
   -1280
   7
   -16
   9
   ```


Figure 5.16: Eastern Comma butterfly (*Polygonia comma*).

   Write a program called `readnum.cpp` that uses a "`while`" loop to read the numbers from `numbers.dat`. Make the program print out the sum of all of the numbers, the value of the largest number, and the value of the smallest number, like this:

   ```
   Sum is -1024
   Largest is 189
   Smallest is -1280
   ```

   Make sure your program does the right thing even if all the numbers are negative.

3. Using *nano*, create a file named `budget.dat` that contains three equal-length columns of numbers, like this:

   ```
   -462.13    486.47    973.79
    755.42    843.04   -963.67
    442.58   -843.02   -462.86
   -233.93   -821.67    399.59
   -379.65   -556.37    837.46
     55.18   -144.93    -93.15
    533.73    804.64    -66.25
   -922.12    914.68   -264.67
   -600.27   -838.59    747.02
   -962.97     49.96   -677.79
   ```

Now write a program named `budget.cpp` that reads this file and adds up the numbers in each column. The program's output should look like this:

```
Column sums are: -1774.16 -105.79 429.47
```

Note that you can limit the number of decimal places you print by using `%.2lf` instead of just `%lf`. This tells printf to print only two numbers after the decimal point.

4. Using *nano*, create the file `grades.dat` shown in Figure 5.12 on Page 154. Now write a program named `meangrade.cpp` that reads `grades.dat` and prints out a list of student IDs along with each student's average grade. Determine the average by adding up the student's grades for the three homework assignments and dividing the result by 3. The program should print `"Student ID"` and `"Mean Grade"` at the top of the output, to tell the user what the numbers mean.

5. Using *nano*, create the file `grades.dat` shown in Figure 5.12 on Page 154. Now write a program named `lowgrade.cpp` that reads `grades.dat` and prints the lowest grade for the first homework assignment, and the ID number of the student who got this grade. Make sure your program tells the user what these numbers mean. (If there's more than one student with the lowest grade, just print the first student ID that has this grade.) Don't assume the grades will always be between zero and 100. (What if the program were given a file full of SAT scores, for example?)

6. Write a program named `oddeven.cpp` that generates 10,000 random integers and sorts them into two files. Put the odd integers into `odd.dat` and the even integers into `even.dat`. Here are a few hints to help you:

   • You can generate a random number with the `rand` function, as we did in Chapter 2. For example:

   ```
   number = rand();
   ```

   • You can use the modulo operator, `%`, to check whether a number is positive or negative. If `number % 2` is zero, then `number` is even. Otherwise it's odd. (Look back at Chapter 4 for more information about the modulo operator.)

   You might find it interesting to look at `odd.dat` and `even.dat` with *gnuplot*. For example, if you start *gnuplot* and give it the command:

   ```
   plot "odd.dat", "even.dat"
   ```


Doing homework.
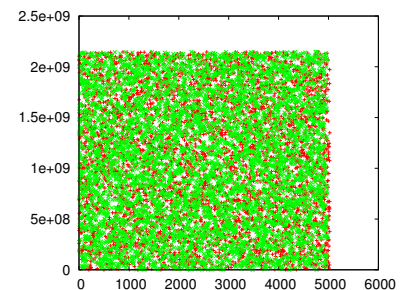*Source: Wikimedia Commons*

Figure 5.17: This is how the data in `odd.dat` and `even.dat` might look if plotted with *gnuplot*.

you should see a rectangle filled with dots of two different colors, one color for odd numbers and the other for even (see Figure 5.17). The extent of the rectangle horiontally will show you how many numbers there are of each type. About half of the numbers you generated should fall into each category, so the rectangle should go up to about 5,000. The vertical axis shows the actual numbers you generated. The height of the rectangle will depend on what kind of C compiler and computer you're using, but it should go up to some very big numbers.

7. Modify Program 5.3 (the "bullet" program) so that it uses a "do-while" loop to track the bullet until it reaches the ground. (See the gray box after bullet program for information about how to do this.) Make the program write out how long (in seconds) it takes the bullet to reach the ground. Call the new program bullettimer.cpp.